AD-A217 580

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

②

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE Oct 3, 1980	3. REPORT TYPE AND DATES COVERED Oct 1-3, 1980
4. TITLE AND SUBTITLE An Automated Program testing methodology and its Implementation		5. FUNDING NUMBERS
6. AUTHOR(S) J.P.Benson and D.M.Andrews		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) General Research Corporation 5383 Hollister Avenue Santa Barbara, Ca 93111		8. PERFORMING ORGANIZATION REPORT NUMBER F49620-79-C-0115
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR Bldg. 410 Bolling AFB, DC 20332		10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFOSR-TR- 90-0147
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words)		
14. SUBJECT TERMS		15. NUMBER OF PAGES 21
		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Internal Memorandum 2315

PRELIMINARY DRAFT

AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION

by

J. P. Benson

D. M. Andrews

Sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, Under Contract F49620-79-C-0115

GENERAL RESEARCH CORPORATION
5383 HOLLISTER AVENUE • PHONE (805) 964-7724
P.O. BOX 6770, SANTA BARBARA, CALIFORNIA 93111

Paper to be presented at 10th International Symposium on Fault Tolerant Computing in Kyoto, Japan, October 1-3, 1980.

AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION

D. M. Andrews and J. P. Benson
General Research Corporation
5383 Hollister Avenue
Santa Barbara, California 93111
1-805-964-7724

ABSTRACT

One of the themes emphasized at recent conferences is that new methods for testing and system development are going to be necessary to keep up with the trends of the future. Specifically cited as challenges for the 80's, were the need to make software less labor intensive and the need for automated programming tools. The testing phase is one area where there are automated tools which subject software to static tests, but there exist few tools which automate the process of testing a program dynamically. Unlike hardware testing where a test pattern may be automatically stepped through and the test results evaluated by comparison with a "gold unit", software has had no similar testing capability. We are just concluding a research effort directed toward rectifying this lack by combining an existing automated testcase generation and evaluation technique with the use of executable assertions to provide a means of automatically assessing the test results. Since the violations of assertions can act as a common denominator to any application, this methodology may be applied to any test object. This method goes one step farther even than the traditional hardware testing methods, because it also has the capability to automatically generate new testcases by perturbing the input values in accordance with an automated "intelligent" evaluation of the past performance of a sequence of inputs.

This paper describes the implementation of this testing method and its use to test a program which computes orbital state vectors from orbital element vectors. The testing of this program required developing assertions for the program, choosing and inserting representative errors into the program, and implementing the search and data collection algorithms for testing.

INTRODUCTION

Developing methods for showing that a computer program is correct has been an active research area in computer science in recent years. One result from this research has been the development of executable assertions. If assertions are used to describe the correct behavior of a program, then they can be used to determine whether or not the program ran correctly. This removes the need of examining the output from the program in detail. The number of assertions which are violated during a test becomes the output of the program. The value of this single output indicates whether the program is operating correctly or not; it also can be mapped as an "error function" which is defined over the input space of the program.

Expressing the correctness of a program in terms of the error function also yields a solution to the problem of generating testcases. It allows standard techniques for maximizing and minimizing functions in multi-dimensional spaces to be applied to the problem of program testing. Automated search techniques such as complex search and heuristic search can be used to find the maximum values of the error function. The input values for which assertions are violated are the input values for which the program fails to work correctly; therefore, it is desirable to find the areas with the maximum violations.

1 PROBLEMS ASSOCIATED WITH TESTING SOFTWARE

Testing has played a rather nebulous part in the development of software. Specifications for the test plan for software systems are often dependent upon the whims and vagaries of the tester. Occasionally the test object may be chosen by size alone, such as, execute the program for a certain length of time or run a large program through the system. The crucial nature of many applications of software at the present time make it imperative to develop a methodology of testing that is general and can be applied to many types of programs, thus avoiding the subjective nature of present testing techniques. One way to eliminate subjectiveness is, of course, to have someone who has not worked on the project do the testing; but this solution in itself brings new problems. One is that extra time must be allowed to bring that person up to speed so that he is familiar enough with the project to be able to intelligently make up testcases and a procedure for testing. Although testing and quality assurance are often a separate department in an organization, it is not commonly the case for software testing.

1.1 Testcases

There are two ways to make testing more secure or thorough: either increase the number of cases or make the cases more specific to the problem. This latter approach requires a lot of human ingenuity in thinking out where the weak spots are and how to test for them. It also contributes to making the cost of testing skyrocket when each set of testcases must be tailor-made to each new set of software.

It is important to choose testcases which uncover errors early in the development cycle, not only because the cost of fixing errors increases dramatically with time,[1] but also because of possible devastating catastrophes that can result from latent errors.

There have been many papers[2-3] on the subject of choosing testcases because it represents one of the most intriguing problems about testing. How does the tester know when enough input data has been chosen to result in meaningful tests? In fault tolerant applications the test data must include not only the usual values of input data but also the unexpected values in case there are intermittent hardware faults. Therefore, it is also necessary to test that the software will function properly when unexpected input values cause the software to reach unexpected states. Software has so many states, an order of magnitude greater than that of the hardware for a given computer, that the number of testcases can be monumental. Even in hardware testing there is a need for fault collapsing because of the large number of input values when there are a large number of input parameters - the number of testcases required increases exponentially.

1.2 Test Results

Software testing is unlike hardware testing because there is no "gold unit" that can be set up to use as a basis for determining if the results of the testing are correct. Unfortunately, test results must be checked manually. In some applications, e.g. ballistic missile defense software, checking test results from one run can take several weeks.[4,5] With software, it is not a matter of determining if a switch is on or off; there is a lot of output to read and analyze.

Last but not least is the psychological aspect of testing that works against ensuring a productive testing phase. Once the software is completed, the programmer is anxious to get onto some other project. The challenging and interesting part is designing and implementing the code, not testing it. No one really wants to find errors in his own code, and, furthermore, checking the output is so tedious that it makes the testing process seem routine and boring.

2 HOW THIS TECHNIQUE ADDRESSES THE PROBLEMS OF TESTING

The major theme that connects most of the problems associated with testing is that of time; it takes time to construct good testcases, time to run them, and time to look at the results. Therefore, one of the ways to address the problem of testing is to automate as much as possible of the testing sequence and to eliminate as much subjectiveness and human intervention as is practical. Fortunately, the basic mechanism to do this, called the Adaptive Tester, has been developed over the past several years in response to the need of the Ballistic Missile Defense Advanced Technology Center to develop tests for complex software. The Adaptive Tester has the following functional components:

1) Machine aids for specification of the testing environment
2) Automatic preparation of initial test cases
3) Automatic performance evaluation
4) Adaptive or learning algorithms for selecting test cases

This present research effort has utilized the components of the Adaptive Tester that generate testcases by automatic perturbation of the input parameters; evaluate past performances of the constructed testcases; and, using this information in a feedback system, generate subsequent testcases. To adapt this powerful capability to this particular application, it was necessary to utilize executable assertions as a means of providing data to the performance evaluator. The executable assertions allow the method to be prescribed in general terms and used for any application, since the only thing that varies from one application to another are the assertions themselves.

2.1 Executable Assertions

Executable assertions can be used to implement complex fault tolerant schemes,[6] but they also have been found very effective as a simple debugging technique and have been utilized extensively in the development of the Software Quality Laboratory (a large verification system). Assertions have been an operational part of the code long enough to permit an impartial assessment of their value. The primary motivation for adding them was to make debugging easier and quicker because the exact statement number of an assertion that is evaluated to "false" during program execution is stated in a message in the output. For example, if the assertion INITIAL (J .GE. 0 .AND. J .LE. MAXJ), is evaluated as false, then it is clear that J is negative or it has exceeded the maximum value for J (MAXJ). Without assertions to direct attention to the parts of the program that are not operating as expected, it is often impossible to find the source of the errors that are causing the problems.

Not only are assertions useful for debugging when new code is being added, but they also have caused latent errors to surface. In addition, the presence of assertions with concomitant FAIL statements which invoke an error processing routine usually prevents premature termination of the execution and allows the program to continue and perform its function.[7-9] Assertions also have proved their worth from the aspect of maintenance and documentation of the system. The code of the Software Quality Laboratory is so voluminous that no one person can be acutely familiar with all parts. The specification in assertions of acceptable ranges of variables helps immensely when new code is being written that will interface with existing codes.

2.2 <u>The Adaptive Tester</u>

The Adaptive Tester has been developed in recent years in response to the need of the Ballistic Missile Defense Advanced Technology Center to develop tests for the software that would simulate actual battle conditions. Devising these tests took an inordinate amount of time because of the number of parameters that could be varied.[4,5] Some way had to be devised which would automatically perturb the parameters. In addition, it required about a month to examine the results of one run, so it was necessary also to be able to automate the assessment of the performance of the software. Various learning algorithms from artificial intelligence were implemented to make a closed-loop testing environment with the capability of handling large quantities of performance data.

2.3 Search Routine

The search routine selected for this experiment is called a complex search.[10-13] The technique was developed by Box for solving for the maximum or minimum of a nonlinear function. The method involves choosing a set of trial test points at random (called a "complex") and determining the performance at each point. The point with the performance farthest from the desired performance boundary is replaced by a point which lies on a line formed by the rejected point and the centroid of the remaining points. A set of coefficients is calculated to determine the exact location of the new point. These coefficients determine the degree of reflection, expansion, shrinkage, contraction, and rotation to be applied in forming the new set of points. This procedure is repeated until a point is computed that matches the desired performance value.

The performance function may have many independent variables, but for the search routine to function correctly, there must be one more point in the complex than the number of independent variables. Figure 1 gives an example of a complex in three dimensions.

Figure 2 shows the effect of each type of operation that can be done on a triangle by the search routine when it finds the worst point (the minimum function value of all the points in the complex) and tries to replace it with a larger function value (the maximum value).

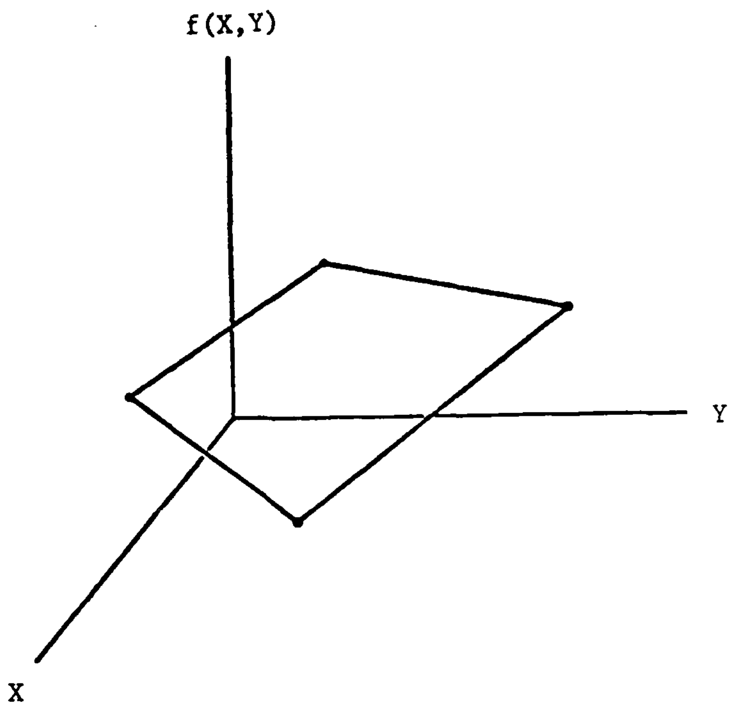

Figure 1. A Complex in Three Dimensions

3 FEATURES OF ADAPTIVE TESTING WITH ASSERTIONS

The testing environment (Figure 3) is very similar to that used for the original Adaptive Testing project. It consists of several sets of software each with a distinct function. First there is the test case construction algorithm which automatically perturbs the input parameters. It uses as input either the original set of test data which has been constructed manually or the output of the search algorithm of the Adaptive Tester. The test object is the program to be tested; it must contain executable assertions. The function of the assertion evaluator is to make a tabulation of the number of assertions violated, including the statement and module number. The information in the test results file is used as input to the search algorithm so it can locate the area of maximum assertion violations and choose the direction of the next test parameter perturbation. Figure 3 shows the feedback system for adaptive testing with assertions where, once the basic test data is initialized manually, the rest of the process is totally automated.

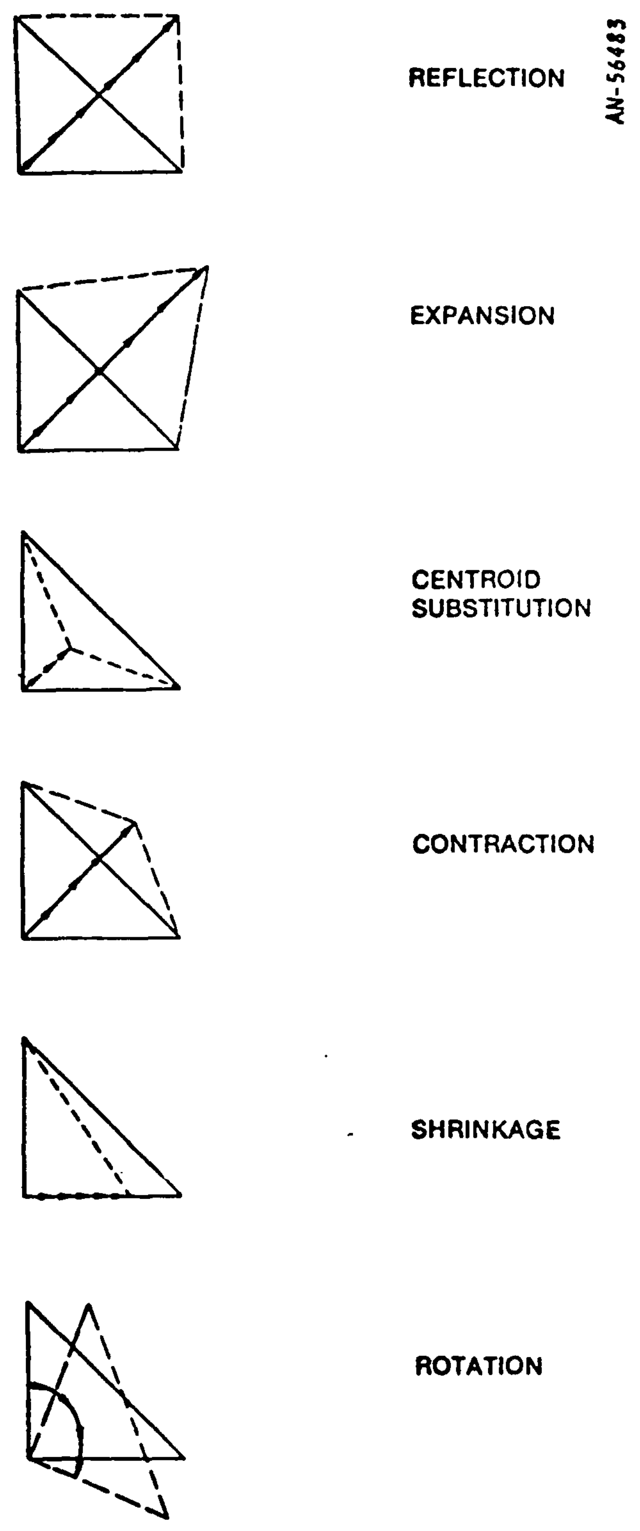

Figure 2. Complex Transformations

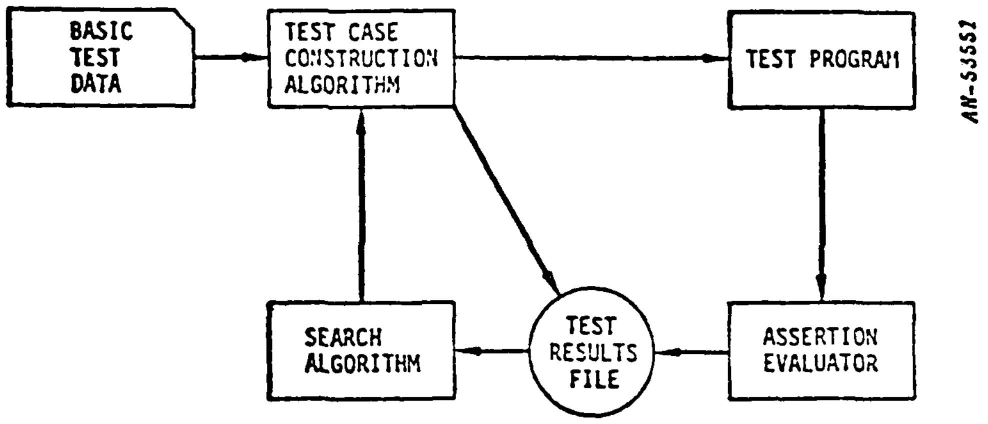

Figure 3. Adaptive Testing With Assertions

4 METHODOLOGY FOR ADAPTIVE TESTING WITH ASSERTIONS

Obviously, if assertions are not in the code already, they must be added; but since they are useful throughout the entire software cycle,[6,9] the optimum way is to incorporate them in the code as it is being written because then they will have been tested as the code itself is tested in smaller sections. In this case, since an already existing program was being tested, it was necessary to write assertions and then to execute the program with the assertions to be sure the assertions themselves were correct. The same thing happened with the second test object as had happened with the first; once the assertions were added to the program, they uncovered latent errors that were completely unsuspected! In most cases, these were errors that only occurred at the boundary conditions.

Some assertions, such as those to check ranges, are simple to write and do not require in-depth familiarity with the algorithm of the code. For example, in a DO loop which has a variable as the upper bound of the index, it is easy to write an assertion which specifies that the value of that variable is greater than zero. More complex and difficult to write are the assertions which provide a check on the results of computations or that express an intricate relationship between the variables. Since it is necessary to have a firm understanding of the program to write these assertions, it is generally best for the person who implements the code to be the one who writes the assertions. The success of this testing technique depends on having a suffient number of assertions which express tight bounds on variables thereby enabling them to detect errors.

Once the assertions have been written, the only other part left in the process requiring human intervention is setting up the first test case. The remaining part of the testing is automated: the performance is evaluated and new testcases generated until a given performance boundary is attained.

5 EXPERIMENTS

Two experiments were performed to determine the usefulness of executable assertions in testing. The purpose of the first experiment was to determine if executable assertions could locate errors; and, if so, what the resulting error space looked like. The first experiment has been described elsewhere.[14] The results of the first experiment indicated that executable assertions were effective in detecting errors; the error function was reasonably well behaved; and the overhead of assertions was ten percent when they are being used with recovery blocks to provide fault tolerance.

The prominent research issues for the second experiment were as follows:

1) Behavior of the error function - Does it confirm the results obtained in the first experiment?

2) Applicable search techniques - Pending determination of the behavior of the error function, what search technique is the most effective in finding errors?

3) Application to large input spaces - What happens when there is a wide stream of input parameters?

The second experiment was more comprehensive since it actually combined the adaptive tester capability with the use of executable assertions. One purpose of this experiment was to provide corroborative evidence of the first experiment. Therefore, instead of continuing with the same program a new test object was selected from a set of routines in the TRAID program library,[15] which is used to compute solutions to orbital mechanics problems. Since the program chosen had been in use for twelve years and had been the test object in another recent experiment,[16] it was assumed to be error free. The function of the program is to take as input an orbit described by a set of eight parameters or orbital elements and produce a state vector representation of a point on the orbit. The state vector includes the time and the position, velocity, and acceleration in three dimensions. The particular point on the orbit is specified by a parameter (MODE), which, in conjunction with another parameter (VALUE), allows the state vector describing the point to be computed.

In this second experiment, three modes of operation were implemented:

Grid -

The values of the input parameters were varied in a uniform set pattern in the form of a grid over the input space. The results from these grid tests were used as a baseline by which to evaluate the search technique.

Search -

Given one initialized value for each of the tested inputs, the search algorithm constructed all subsequent test cases.

Grid and Search -

Instead of having the initial points on the complex being constructed of random testcases, a set of twelve values for each of the tested inputs were given as input to the search algorithm (these values were derived by sorting on the number of assertion violations obtained in the results from running the grid tests; the values of the inputs associated with the highest number of violations were passed to the search routine).

In each mode of operation, three variables were varied: MODE, VALUE, and the eccentricity of the orbit; but for the search mode, additional tests were run in which all the input parameters were allowed to vary. The standard orbit was input to the test driver program which then determined the mode of operation for the test. The data collection routines recorded the number of assertions violated in each test along with the values of the input variables.

A test driver was written to interface the search routine with the test program and initialize the first test. It also initializes the values of all variables needed to conduct the test and reads in the basic set of orbital parameters which are common to all tests. It reads the values of the variables to be varied and their ranges and, for the grid test, divides the ranges up into intervals and selects a set of values for each variable corresponding to this division. It also calculates the dependent orbital parameters and runs the grid tests. The search routine itself runs the search tests when the system is run in this mode.

Errors were generated for the test program using a procedure developed by Brooks.[17] The method uses error types and frequencies from a previous study[18] to randomly select a set of errors to be "seeded" in the program. Some types of errors were not chosen for the study, such as documentation, data definition, etc., because the experiment was specifically concerned with detecting run-time errors. The types of errors used were computational errors, logic errors, data handling errors, and interface errors. In generating errors for the experiment, statement types and other descriptive information about the test program were generated automatically using an automated program verification system, SQLAB. Each statement in the program was classified by type, and a table matching the error catagories to statement types was constructed. From a list of available error sites, potential error sites were randomly selected and matched with the error categories. Once the assertions were written and checked out, errors were introduced one at a time to determine how effective this technique was in detecting errors.

For each error, a grid test was run and then tests using the automated search technique were run to see if the results were the same. Testing was done in two ways using the search strategy: first by varying MODE, VALUE, and one other variable; and then by varying all of the variables in the orbit. The search routine was allowed to run until it found a preset number of assertion violations (representing the performance value); then this number was automatically stepped up by one and the search algorithm tried to find another combination of input values which would cause the new number of violations to occur. In this way, the performance value was maximized. The testing process was arbitrarily set to terminate when one hundred tests were run, but each test actually consisted of several subtests because the values of MODE and VALUE were varied within each test. The report that is produced at the conclusion of the runs is shown in Figure 4; in this test MODE, VALUE, and one other variable, ORBIT(6) - the eccentricity - were varied.

The results of the experiment demonstrated the effectiveness of the assertions in detecting errors. Of the original 24 errors, nine (thirty-eight percent) were detected by original assertions, and eight (thirty-three percent) were detected by assertions that were added after the testing began. There were seven errors that could not be detected by assertions. Two of the errors could not be detected by this testing method because they existed in sections of code which were only traversed when an error had occurred. Most of the remaining errors could have been detected through the use of static-analysis tools which test consistency of the variables. Each of these errors and the reason for it's not being detected is listed in Table 1.

For all but four of the errors, the search methods detected the same errors as the grid tests; but they were able to do so much more efficiently and used much less computer time.

Table 2 shows the efficiency of the search technique when all variables are varied; it lists the number of the test in which the first assertion violation was detected. Fifteen of the seventeen detectable errors were detected within the first seven tests devised by the search technique. The grid technique was initially run for 317 tests and discovered all but one of the detectable errors; but 683 tests had to be run to detect error number 52.

```
                 ********** FINAL REPORT **********

  #RUN      INPUT1          #FALSE      #DIFFERENT  MODE     VALUE
                            ASSERTION    ASSERTION

     7     .7526              2            2         4      2477545.659
     9     .6048              2            2         5      9849931.060
    12     .2700              2            2         4      13958923.49
    13     .9000              1            1         5      24389119.03
    24     .2899              2            2         4      8871067.739
    25     .3879              2            2         5      1760571.330
    30     .2910              2            2         5      20758872.74
    34     .7346              1            1         4      22330022.80
    35     .1852              2            2         5      27015515.91
    37     .3555              2            2         4      19513234.41
    44     .6973              2            2         4      4044234.171
    45     .6235              2            2         5     0.
    47     .5851              2            2         4      4533190.345
    49     .9000              2            2         5     0.
    51     .7234              2            2         4      4533190.345
    53     .9000              2            2         5      5737662.000
    55     .7234              2            2         4      7402021.345
    63     .7053              2            2         5     0.
    65     .6261              2            2         4      4533190.345
    73     .7474              2            2         5     0.
    75     .6471              2            2         4      4533190.345
    83     .8071              2            2         5     0.
    85     .6769              2            2         4      4533190.345
    86     .1774              1            1         4      23124989.06
    87     .9000              2            2         5      1415222.244
    89     .7228              2            2         4      5588024.749
    90     .9000              2            2         5      1939816.571
    91     .1557              2            2         4      6107643.223
    92     .5503              2            2         4      9951144.293
    93     .3530              2            2         4      8029393.758
    94     .4955              2            2         4      11674092.79
    95     .8433              1            1         5      18860857.79
    96     .5648              2            2         4      11115483.73
    97     .7040              1            1         4      14988170.76
    98     .6108              1            1         4      17226371.99
    99     .2554              2            2         5      3876077.474
   100     .5485              2            2         4      11161715.66
   101     .4019              2            2         5      7518896.567
   102     .1000              2            2         5      7331062.939

                    INPUT1 =      ORBIT(6)

            MODULE     STMT#       TYPE          FAILURES*

            ORBP       109        ASSERT            34
            OUTCHK     142        ASSERT            38

            * HOW MANY RUNS EACH ASSERTION FAILED IN     102   RUNS
```

Figure 4. Summary of Search Testing for Error 13

TABLE 1

ERRORS NOT DETECTED BY ASSERTIONS

Description	Reason For Not Being Detected
Variables assigned values in incorrect order	An error must occur for this section of code to be executed
Test and branch statement deleted	Checks for out of range input values
Variable name mispelled in computed goto	Difficult to state an assertion for this error
Data statement deleted	Fortran compiler initializes all variables to zero
Real variable declared as integer	Difficult to state an assertion for this error
Subroutine call out of place	An error must occur for the section of code to be executed
Wrong number of arguments in subroutine call	Difficult to state an assertion for this error

TABLE 2

DETECTION OF ASSERTION VIOLATIONS BY SEARCH METHOD

Error Number	Test Number of First Assertion Violation
1	5
3	2
13	7
14	5
28	*
31	4
37	5
41	3
47	57
48	3
52	3
54	3
56	5
57	7
64	2
67	5
74	2

* No assertion violations detected.

CONCLUSION

The results from this experiment indicate that this automated testing technique has the potential for finding errors (logic, computational, etc.) that are difficult, even impossible to find in other ways. In addition, use of the search algorithm completely eliminates the subjectiveness in constructing testcases and broadens the testcase base. By automating so many facets of the testing process, the cost of testing can be reduced dramatically.

Another benefit of this project is a further merging of the testing process with the requirements of fault tolerant computing, since assertions can be used to detect errors during system operation and to implement methods of recovery from software and hardware errors.[6] In fault tolerant applications requiring minimal overhead, it may even be possible to optimize assertion coverage through the use of this testing technique.

Although the emphasis in this paper has been on the effectiveness of executable assertions in combination with an adaptive search algorithm to provide testing, this is more than just a new and innovative way to test software. It is a response to the needs of the future which require more automation of the software development process and more accurate testing environments to provide software reliability.

ACKNOWLEDGEMENT

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under Contract F49620-79-C-0115. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

REFERENCES

1 Tomlinson G. Rauscher, "A Unified Approach to Microcomputer Software Development", Computer Magazine, June 1978.

2 John B. Goodenough, Susan L. Gerhart, "Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

3 W. E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol SE-4, July 1978.

4 D. W. Cooper, "Adaptive Testing," Second International Conference on Software Engineering, 13-15 October 1976, San Francisco, CA.

5 D. W. Cooper, Adaptive Learning Requirements and Critical Issues, General Research Corporation CR-4-708, January 1977.

6 Dorothy Andrews, "Using Executable Assertions for Testing and Fault Tolerance," 1979 International Conference on Fault Tolerant Computing, Madison, Wisconsin, June 20-22,1979.

7 Sabina Saib, "Distributed Architectures for Reliability," Proceedings of the AIAA Computers in Aerospace Conference II, Los Angeles, October 1979.

8 Sabina Saib, "Verification and Validation of Avionics Simulation," Proceedings of the AGARD Avionics Panel on Modeling and Simulation of Avionics and Command, Control and Communications Systems, Paris, France, October 1979.

9 Dorothy Andrews, "Using Executable Assertions for Testing," Proceedings of the 13th Annual Asilomar Conference on Circuits, Systems and Computers, November 1979.

10 M. J. Box, "A New Method of Constrained Optimization and a Comparison with Other Methods," Computer Journal, Vol. 8 (1965).

11 J. A. Richardson and J. L. Juester, "Algorithm 454--The Complex Method for Constrained Optimization", Comm. ACM, Vol. 6, No. 8, August 1973.

12 K. D. Shere, "Remark on Algorithm 454," Comm. ACM, Vol. 7, No. 8, August 1974.

13 K. D. Shere, The Box Optimization Method, Naval Ordnance Laboratory NOLTR-74-167, October 25, 1974.

14 J. Benson, A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February 1980.

15 T. Plambeck, The Compleat Traidsman, General Research Corporation IM-711/2, revised edition, September 1969.

16 R. N. Meeson, C. Gannon, "An Empirical Evaluation of Static Analysis and Path Testing," Proceedings of AIAA Computers in Aerospace Conference II, Los Angeles, October 1979.

17 N. B. Brooks, An Experimental Evaluation of Software Testing General Research Corporation CR-1-854, May 1979.

18 T. A. Thayer, et al., Software Reliability Study, TRW Defense and Space Systems Group RADC-TR-76-238, Redondo Beach, California, August 1976.